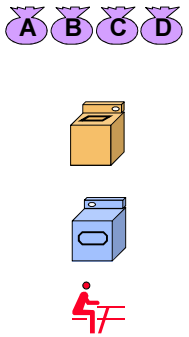
计算机组成原理

计算机组成原理课程组
（刘旭东、高小鹏、肖利民、牛建伟、栾钟治）

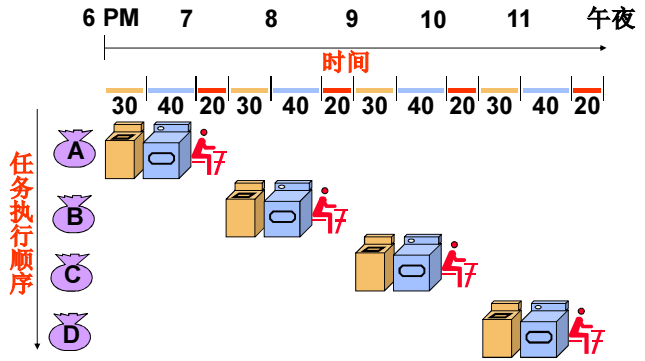
第六讲 MIPS处理器设计

洗衣房：生活中的流水线

- 处理器：洗衣房
- 4条指令：张三、李四、王五、周大麻子
  - 指令：洗衣服
  - 指令过程：洗涤→烘干→熨整
- 指令各阶段延迟
  - 洗衣：30分钟
  - 烘干：40分钟
  - 熨整：20分钟

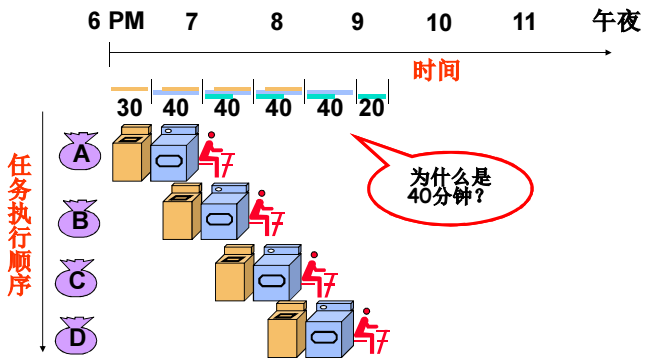


串行洗衣房



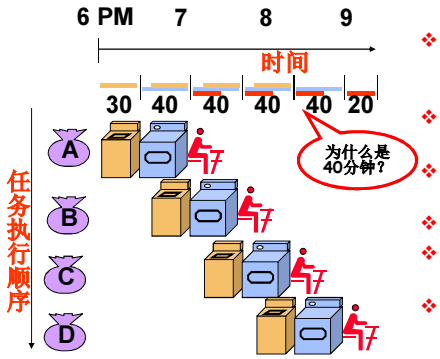
- 串行洗衣房：6个小时完成4个任务
- 如果采用流水线技术，那么。。。

流水化洗衣房：尽可能早的开始工作



- 流水化洗衣房：3.5小时完成4个任务

流水线性质

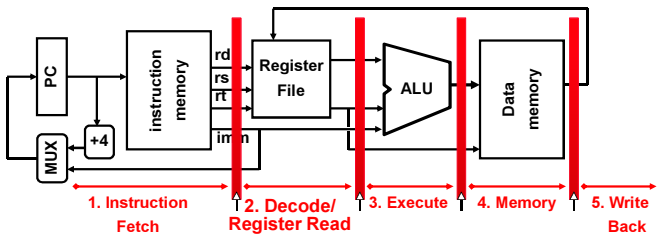


- 流水线不改善单个任务处理延迟(latency)，但改善了整体工作负载的吞吐率
- 流水线速率受限于最慢的流水段
- 多个任务同时工作，但占用不同的资源
- 潜在加速比 = 流水线级数
- 流水段执行时间不平衡，则加速比下降
- 填充流水线和排放流水线，加速比下降

MIPS数据通路的5个阶段

1) IF: 取指令（Instruction Fetch），PC值变化
2) ID: 指令译码（Instruction Decode），读寄存器
3) EX: 执行运算（Execution），ALU操作
   Load/Store: 计算地址
   其他指令: 执行运算操作
4) MEM: 访存
   Load: 从Memory中读取数字
   Store: 将数据写入Memory
5) WB: 数据写回寄存器（Write Data Back to Register）

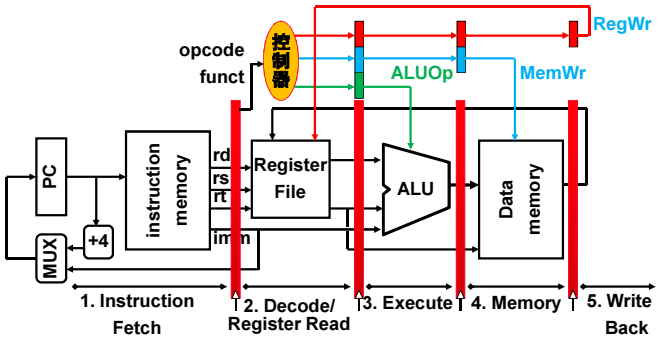
流水线数据通路



- 在不同阶段之间增加寄存器
  - 保存前一个周期产生的信息
- 5 阶段流水线
  - 意味着时钟频率实际上变为原来的5倍

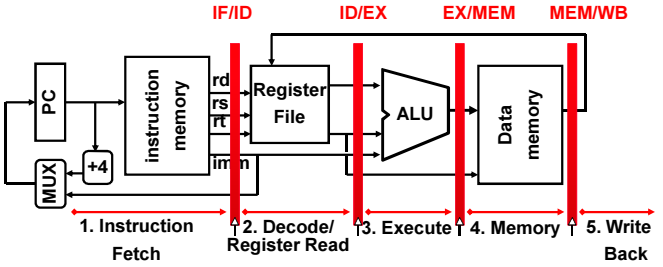
流水的控制信号

- 控制器：译码产生控制信号，与单周期完全相同
- 控制信号流水寄存器：控制信号在寄存器中传递，直至不再需要



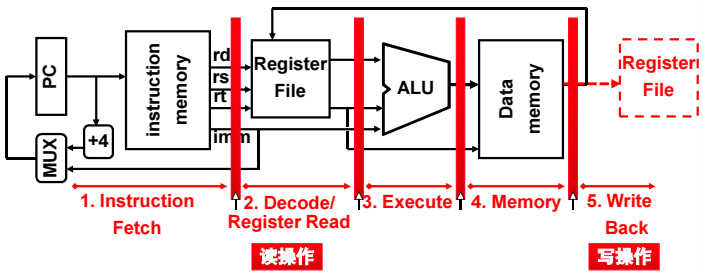
正确认识流水线—流水线寄存器

- 命名法则：前级/后级
  - 示例：IF/ID，前级为读取指令，后级为指令译码(及读操作数)
- 功能：时钟上升沿到来时，保存前级结果；之后输出至下级组合逻辑
  - 也可能直接连接到下级流水线寄存器
    - 例如ID/EX保存的从RF读出的第2个寄存器值，就直接传递到EX/MEM

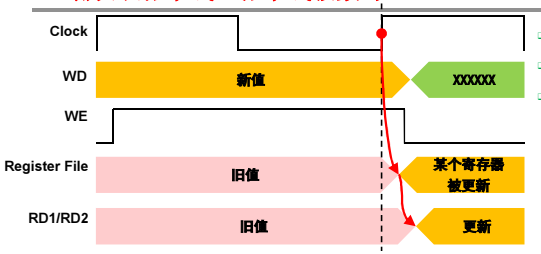


正确认识流水线—流水线级数与RF

- N级流水线：必须有N级流水线寄存器
  - 插入N-1级流水线寄存器，最后一级为Register File
- RF：这是一个特殊部件，有2次使用
  - 读：第2级；写：第5级

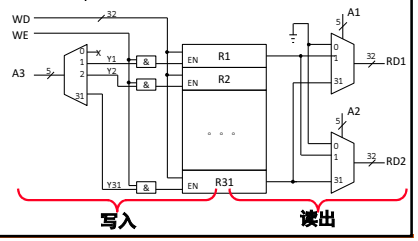


正确认识流水线：流水线级数与RF



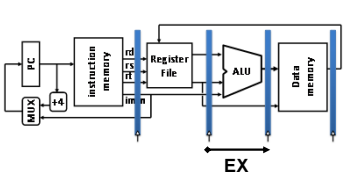
- RF具有2重行为
- 读出：组合逻辑
- 写入：时序逻辑(寄存器)

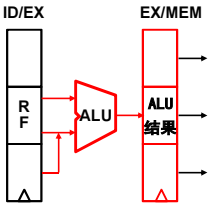
- RF：上升沿被更新
  - WE有效，则WD被写入A3
- 输出：与时钟无关，仅与RF内容及A1/A2相关



正确认识流水线：流水阶段与流水线寄存器

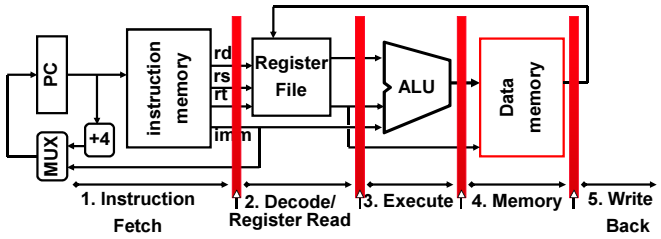
- 流水阶段：组合逻辑+寄存器
  - 起始：前级流水线寄存器的输出
  - 中间：组合逻辑（如ALU）
  - 结束：写入后级流水线寄存器
  - 当时钟上升沿到来时，组合逻辑计算结果存入后级寄存器
- 示例：EX阶段
  - 起始：ID/EX流水线寄存器中的RF寄存器/扩展单元的输出
  - 中间(组合逻辑)：ALU完成计算
  - 结束(寄存器)：在clock上升沿到来时，结果写入EX/MEM中相应寄存器





正确认识流水线：DM

- 写入时：表现为寄存器，属于MEM/WB寄存器范畴
- 读出时：可以等价为组合逻辑
  - 与RF的读出是类似的

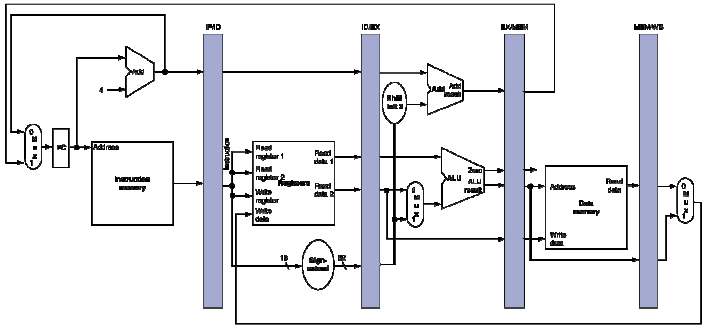


流水线的变化

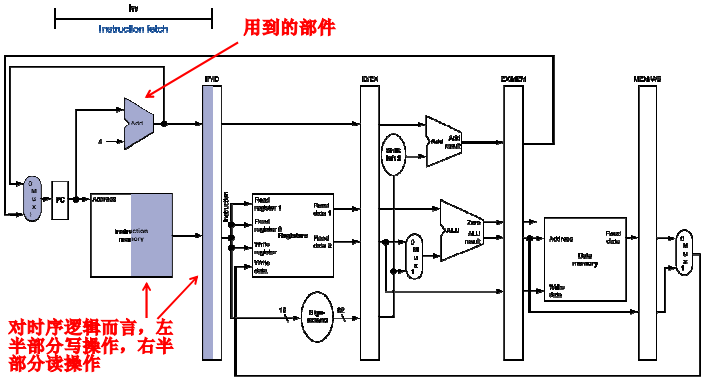
- 寄存器影响信息的流动
  - 名字寄存器(例如：IF/ID，标明相邻的阶段)
  - 寄存器*分隔*各阶段之间的信息流
  - 在任何的时间片段，每一个阶段执行着*不同的*指令！
- 需要重新验证数据通路（连线和部件的放置）

流水线的细节分析

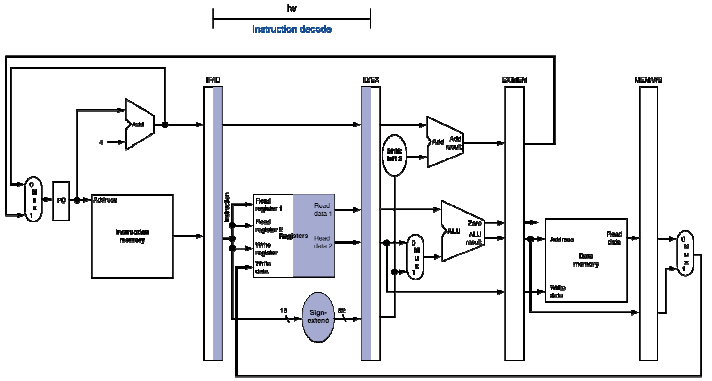
- 验证lw指令在流水线中的流动



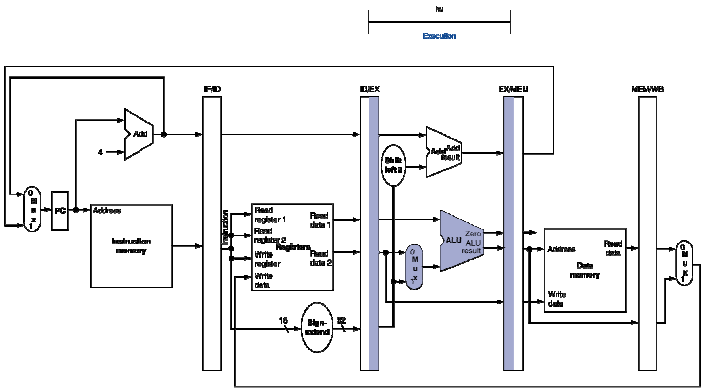
Lw指令的取值（IF）



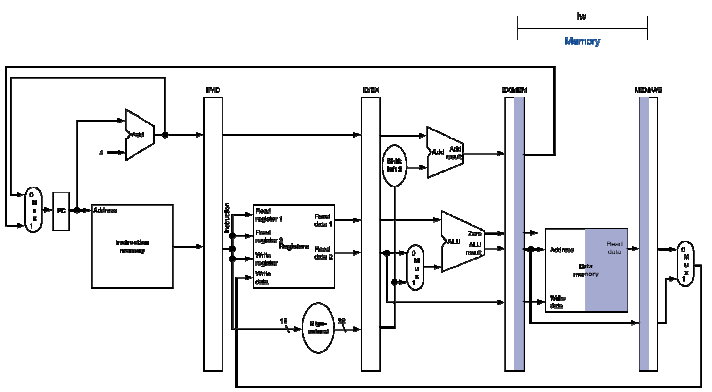
Lw指令的译码（ID）



Lw指令的执行（EX）

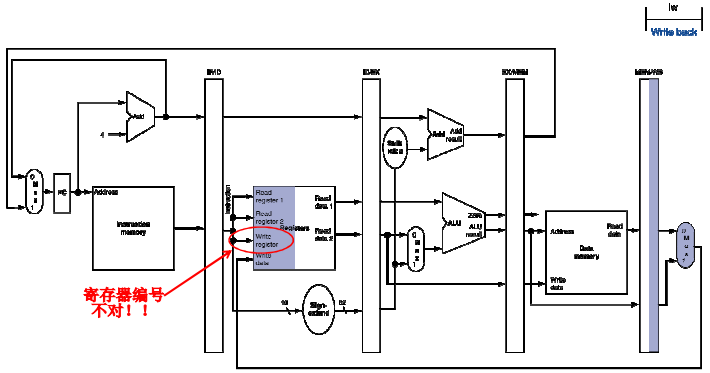


Lw指令的访存（MEM）



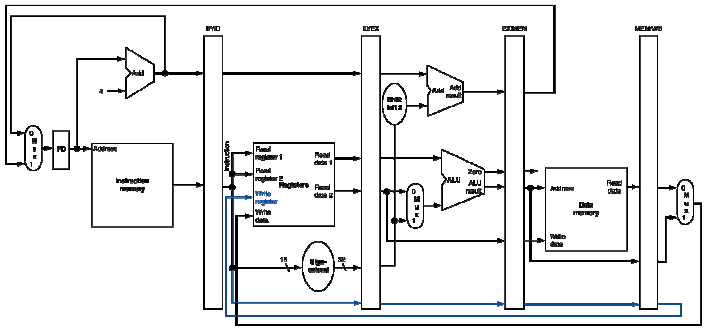
Lw指令的写回（WB）

- 这里有些不对头！！

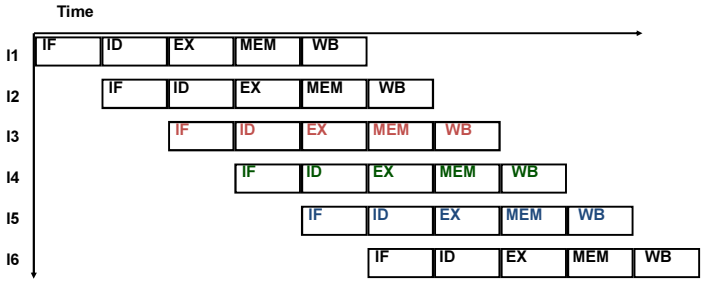


修正数据通路

- 这样，任何指令写寄存器就不会有问题了！



流水线执行的表示



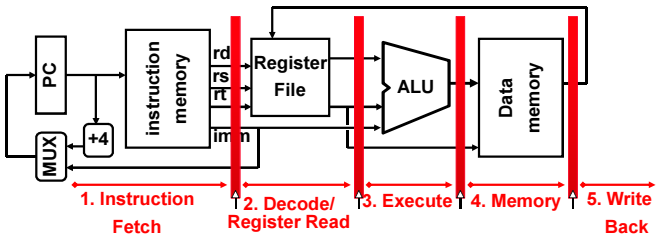
- 所有的指令必须有同样的阶段数，所以有些指令在某些阶段会处于空闲状态（idle）
  - 例如MEM阶段对所有算术运算指令

时钟驱动的流水线时空图

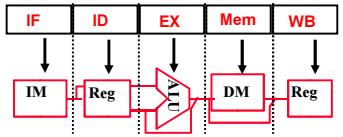
- 用途：精确分析指令/时间/流水线3者关系时
  - 行：某个时钟，指令流分别处于哪些阶段
  - 列：某个部件，在时间方向上的执行了哪些指令
- 注意区分流水阶段与流水线寄存器的关系
- 可以看出，在clk5后，流水线全部充满
  - 所有部件都在执行指令
    - 只是不同的指令

				IF级	ID/RF级	EX级	MEM级	WB级
相对PC地址偏移	指令	CLK	PC	IM	IF/ID	ID/EX	EX/MEM	MEM/WB	RF
0	Instr 1	⮥ 1	0→4	Instr 1	Instr 1				
4	Instr 2	⮥ 2	4→8	Instr 2	Instr 2	Instr 1			
8	Instr 3	⮥ 3	12→12	Instr 3	Instr 3	Instr 2	Instr 1		
12	Instr 4	⮥ 4	12→16	Instr 4	Instr 4	Instr 3	Instr 2	Instr 1	
16	Instr 5	⮥ 5	16→20	Instr 5	Instr 5	Instr 4	Instr 3	Instr 2	Instr1
20	Instr 6	⮥ 5	16→20	Instr6	Instr6	Instr5	Instr4	Instr3	Instr2

流水线图

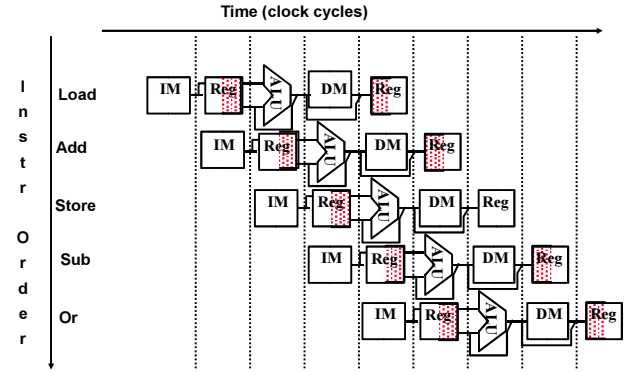


❖使用数据通路图表达流水线



流水线的图形化表示

- 寄存器堆: 右半部读, 左半部写



指令级并行(ILP)

❖流水线允许使用相同的部件同时执行多条指令的某个部分

➢ *instruction level parallelism*

流水线的性能

❖假设每个阶段的时间如下

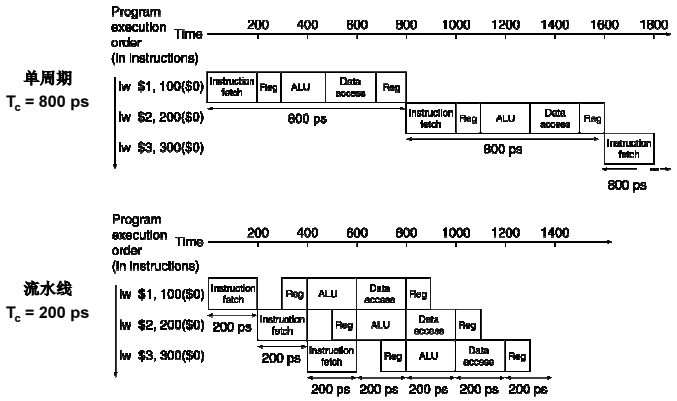
➢寄存器读/写100ps

➢其他阶段200ps

Instr	Instr fetch	Register read	ALU op	Memory access	Register write	Total time
lw	200ps	100 ps	200ps	200ps	100 ps	800ps
sw	200ps	100 ps	200ps	200ps		700ps
R-format	200ps	100 ps	200ps		100 ps	600ps
beq	200ps	100 ps	200ps			500ps

❖流水线的时钟频率?

➢对比流水线数据通路和单周期数据通路



流水线的加速比

- 使用T_c (完成一条指令的时间) 计算加速比

$$T_{c,pipelined} \geq \frac{T_{c,single-cycle}}{\text{Number of stages}}$$

  - 各阶段均衡的时候取等号 (各阶段消耗相等的时间)
- 如果不均衡则加速比会下降
- 加速比的获得是由于增加了吞吐量
  - 每条指令的延迟并没有减少

流水线和ISA设计

- MIPS的指令集就是为流水线设计的！
- 所有的指令都是32-bits
  - 方便在一个周期内完成取指和译码
- 指令格式简单规范, 2个源操作数域的位置固定不变
  - 能够在一步内译码和读寄存器
- 只有Load和Store指令操作Memory
  - 能够在第三阶段计算地址，第四阶段访存
- 内存对齐
  - 访存只需一个周期

流水线冒险

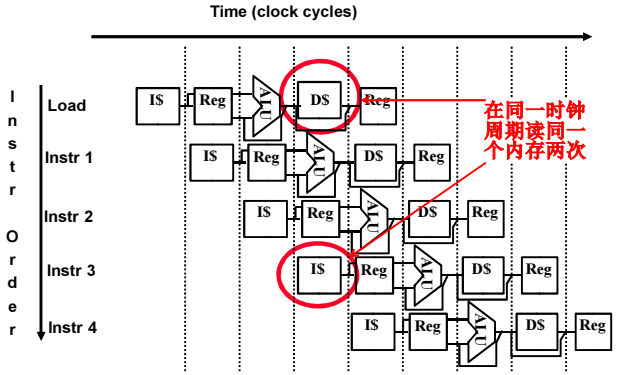
在下一个时钟周期妨碍下一条指令执行的状况

1) *结构冒险*
   - 某个需要的资源忙 (比如在多个阶段都要用到)
2) *数据冒险*
   - 指令之间的数据依赖
   - 需要等待之前的指令以完成数据读写
3) *控制冒险*
   - 执行流依赖于之前的指令

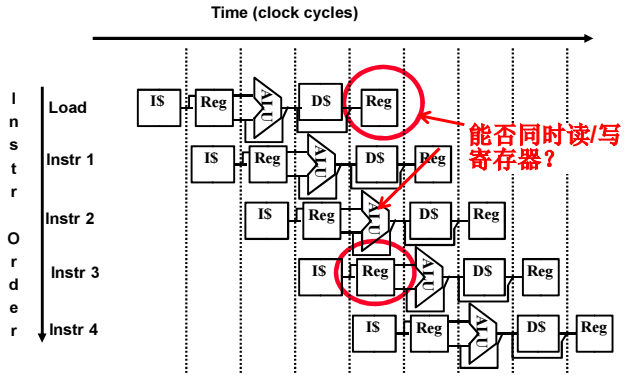
结构冒险

- 对资源使用的冲突
- 只使用一个memory的MIPS流水线
  - Load/Store需要访问内存
  - 取指令可能不得不暂停相应的周期
    - 产生一个流水线“气泡”
- 因此, 流水线数据通路需要单独的指令和数据存储器
  - 单独的 L1 I$ 和 L1 D$

结构冒险#1: 单一存储器

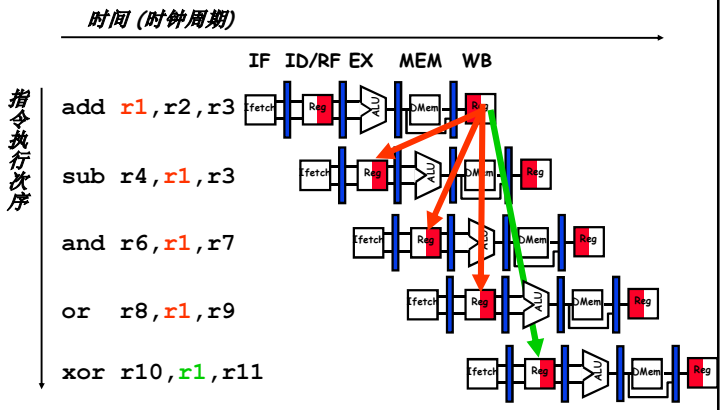


结构冒险#2: 寄存器

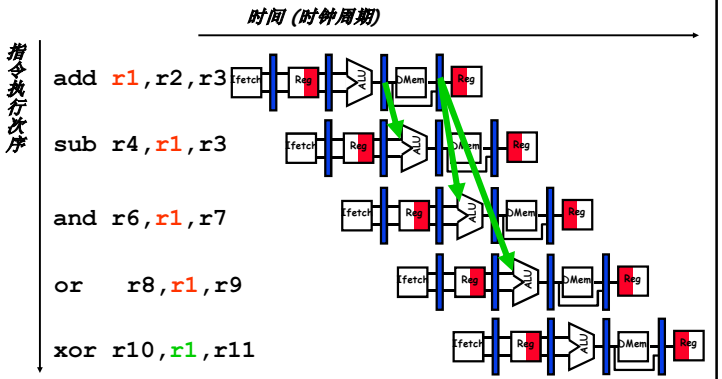


- 两种可能的解决方案:
  1) 分割寄存器堆的访问周期: 时钟周期的前半段写，后半段读
     - 可行，因为寄存器堆的访问速度非常快 (小于ALU阶段所需时间的一半)
  2) 构建具有独立读/写端口的寄存器堆
- 结论: 寄存器读/写可以在同一个时钟周期进行

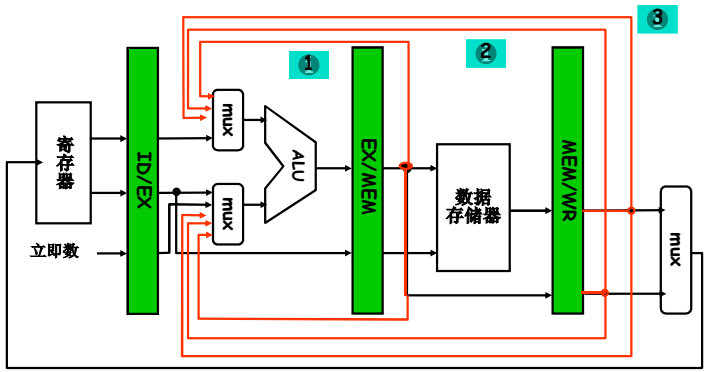
数据冒险



数据冒险解决策略—旁路

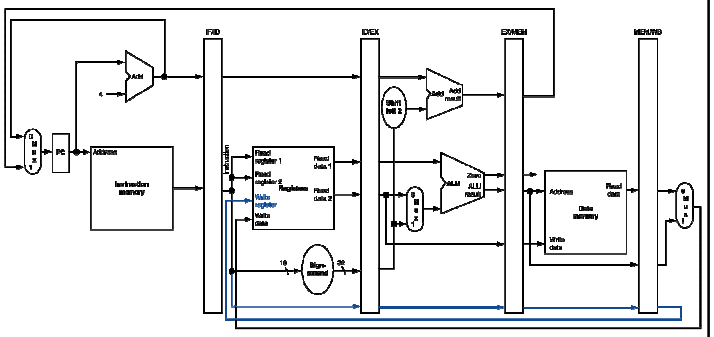


调整硬件结构支持旁路

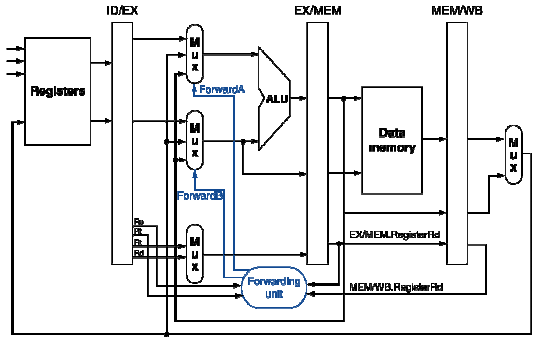


带转发的数据通路

- 需要改变什么？

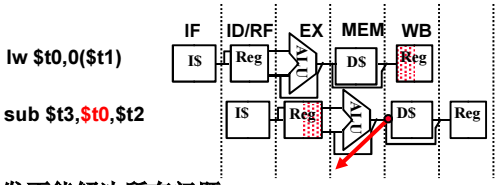


❖增加转发单元



数据冒险：Loads

❖数据流回到从前会带来风险

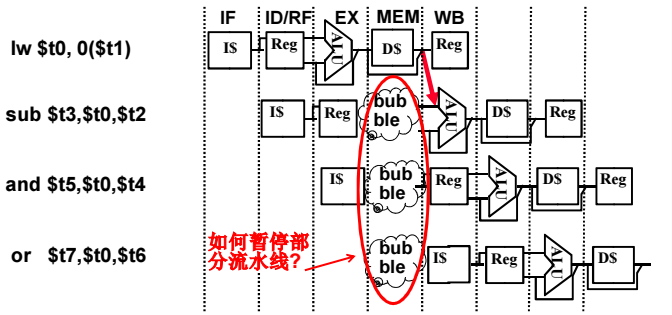


- 靠转发不能解决所有问题
  - 必须暂停依赖于load的指令，然后转发（需要更多硬件）

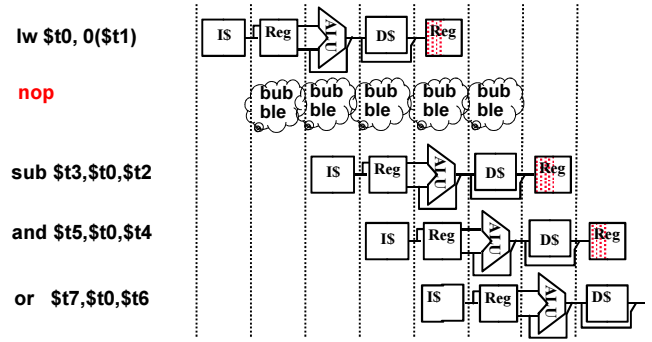
❖硬件暂停流水线

- “硬件互锁”

直观上看，这正是我们想要的，但是实际当中的暂停是“水平”实现的



❖暂停等价于nop



load导致的数据冒险：Clk1上升沿后

- 指令流
  - lw进入IF/ID
  - PC：指向sub指令的地址
    - PC ← PC + 4
  - IM：输出sub指令

后续不再分析PC和IM

		IF级	ID级	EX级	MEM级	WB级			
地址	指令	CLK	PC	IM	IF/ID	ID/EX	EX/MEM	MEM/WB	RF
0	lw $t0, 0($t1)	↑ 1	0→4	lw→sub	lw				
4	sub $t3, $t0, $t2								
8	and $t5, $t0, $t4								
12	or $t7, $t0, $t6								
16	add $t1, $t2, $t3								

load导致的数据冒险：Clk2上升沿后

- 指令流
  - sub进入IF/ID寄存器；lw进入ID/EX寄存器
- 冲突分析：冲突出现
- 执行动作：设置控制信号，在clk3插入nop指令
  - ①冻结IF/ID：sub继续被保存
  - ②清除ID/EX：指令全为0，等价于插入NOP
  - ③禁止PC：防止PC继续计数，PC应保持为PC+4

		IF级	ID级	EX级	MEM级	WB级			
地址	指令	CLK	PC	IM	IF/ID	ID/EX	EX/MEM	MEM/WB	RF
0	lw $t0, 0($t1)	↑ 1	0→4	lw→sub	**lw**				
4	sub $t3, $t0, $t2	↑ 2	4→8	sub→and	**sub**	**lw**			
8	and $t5, $t0, $t4								
12	or $t7, $t0, $t6								
16	add $t1, $t2, $t3								

load导致的数据冒险：Clk3上升沿后

- 指令流
  - sub进入ID/EX；lw进入EX/MEM
  - ID/EX向ALU提供数据
- 冲突分析：冲突解除
  - 转发机制将在clk4时可以发挥作用

		IF级	ID级	EX级	MEM级	WB级			
地址	指令	CLK	PC	IM	IF/ID	ID/EX	EX/MEM	MEM/WB	RF
0	lw $t0, 0($t1)	↑ 1	0→4	lw→sub	**lw**				
4	sub $t3, $t0, $t2	↑ 2	4→8	sub→and	**sub**	**lw**			
8	and $t5, $t0, $t4	↑ 3	8→8	and	sub	nop	lw		
12	or $t7, $t0, $t6								
16	add $t1, $t2, $t3								

load导致的数据冒险：Clk4上升沿后

- 指令流
  - lw：结果存入MEM/WB。
  - sub：进入ID/EX。故ALU的操作数可以从MEM/WB转发
- 执行动作
  - 控制MUX，使得MEM/WB输入到ALU

		IF级	ID级	EX级	MEM级	WB级			
地址	指令	CLK	PC	IM	IF/ID	ID/EX	EX/MEM	MEM/WB	RF
0	lw $t0, 0($t1)	↑ 1	0→4	lw→sub	**lw**				
4	sub $t3, $t0, $t2	↑ 2	4→8	sub→and	**sub**	**lw**			
8	and $t5, $t0, $t4	↑ 3	8→8	and	sub	nop	lw		
12	or $t7, $t0, $t6	↑ 4	8→12	and→or	**and**	sub	nop	lw结果	
16	add $t1, $t2, $t3								

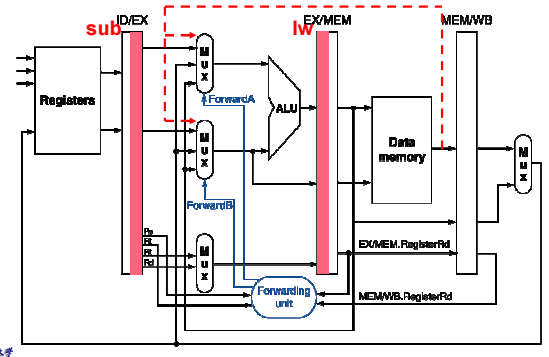
load导致的数据冒险：Clk5上升沿后

- 指令流
  - lw：结果回写至RF
  - sub：结果保存在EX/MEM

			IF级	ID级	EX级	MEM级	WB级	
地址	指令	CLK	PC	IM	IF/ID	ID/EX	EX/MEM	MEM/WB	RF
0	lw $t0, 0($t1)	1	0→4	lw→sub	lw				
4	sub $t3, $t0, $t2	2	4→8	sub→and	sub	lw			
8	and $t5, $t0, $t4	3	8→8	and	sub	nop	lw		
12	or $t7, $t0, $t6	4	8→12	and→or	and	sub	nop	lw结果	
16	add $t1, $t2, $t3	5	12→16	or→add	or	and	sub结果	nop	lw结果

load导致的数据冒险

- Q：如果设置从DM到ALU输入的转发，这个设计优劣如何？
  - 设计初衷：将DM读出数据提前1个clock转发至ALU，从而消除lw指令导致的数据相关，无需插入NOP

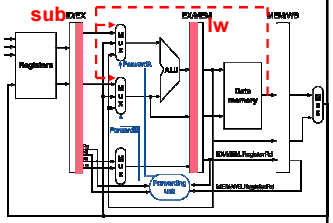


load导致的数据冒险

- A：功能虽然正确，但CPU时钟频率大幅度降低
  - 原设计：*f* = 5GHz
    - 各阶段最大延迟为200ps
  - 新设计：*f* = 2.5GHz
    - EX阶段$_{修改后}$= ALU延迟 + DM延迟 = 400ps
    - EX阶段延迟成为最大延迟

警惕：木桶原理！

流水线各阶段延迟不均衡，将导致流水线性能严重下降



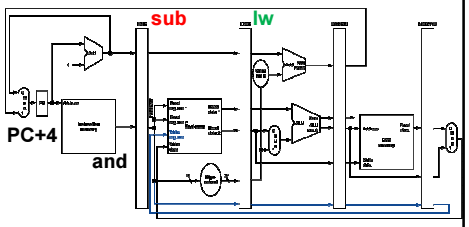
前面PPT的数据

Instr fetch	Register read	ALU op	Memory access	Register write
200ps	100 ps	200ps	200ps	100 ps

如何插入NOP指令？

- 检测条件：IF/ID的前序是lw指令，并且lw的rt寄存器与IF/ID的rs或rt相同
- 执行动作：
  - ①冻结IF/ID：sub继续被保存
  - ②清除ID/EX：指令全为0，等价于插入NOP
  - ③禁止PC：防止PC继续计数，PC应保持为PC+4

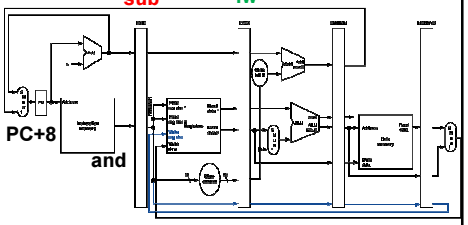
地址	指令
0	lw $t0, 0($t1)
4	sub $t3, $t0, $t2
8	and $t5, $t0, $t4
12	or $t7, $t0, $t6
16	add $t1, $t2, $t3



如何插入NOP指令？

Cycle N

sub lw

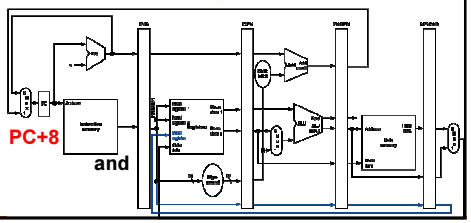


地址 指令

```
 0  lw  $t0, 0($t1)
 4  sub $t3, $t0, $t2
 8  and $t5, $t0, $t4
12  or  $t7, $t0, $t6
16  add $t1, $t2, $t3
```

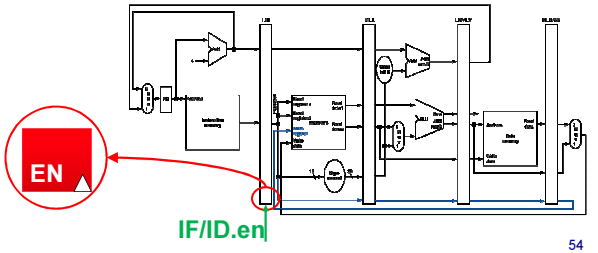
Cycle N+1

sub nop/bubble lw



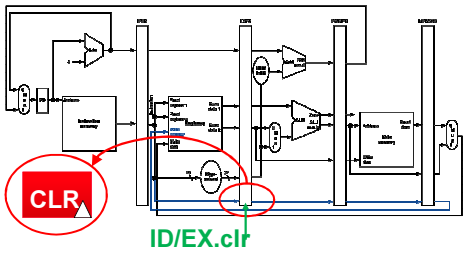
如何插入NOP指令？

- 执行动作：
  - ①冻结IF/ID：sub继续被保存
  - ②清除ID/EX：指令全为0，等价于插入NOP
  - ③禁止PC：防止PC继续计数，PC应保持为PC+4
- 数据通路：将IF/ID修改为使能型寄存器
- 控制系统：增加IF/ID.en控制信号
  - 当IF/ID.en为0时，IF/ID在下个clock上升沿到来时保持不变



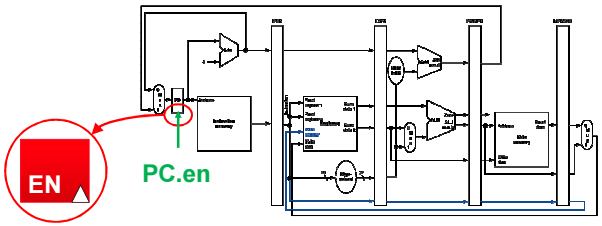
如何插入NOP指令？

- 执行动作：
  - ①冻结IF/ID：sub继续被保存
  - ②清除ID/EX：指令全为0，等价于插入NOP
  - ③禁止PC：防止PC继续计数，PC应保持为PC+4
- 数据通路：将ID/EX修改为复位型寄存器
- 控制系统：增加ID/EX.clr控制信号
  - 当ID/EX.clr为0时，ID/EX在下个clock上升沿到来时被清除为0



如何插入NOP指令？

- 执行动作：
  - ①冻结IF/ID：sub继续被保存
  - ②清除ID/EX：指令全为0，等价于插入NOP
  - ③禁止PC：防止PC继续计数，PC应保持为PC+4
- 数据通路：将PC修改为使能型寄存器
- 控制系统：增加PC.en控制信号
  - 当PC.en为0时，PC在下个clock上升沿到来时保持不变



如何插入NOP指令？

- lw冒险处理示例伪代码
- 注意：时序关系
  - 各信号在clk2上升沿后有效
  - NOP是在clk3上升沿后发生，即寄存器值在clk3上升沿到来时发生变化(或保持不变)

```
if (ID/EX.MemRead) &
   ((ID/EX.rt == IF/ID.rs) |
    (ID/EX.rt == IF/ID.rt)
    IF/ID.en ← 禁止
    ID/EX.clr ← 清除
    PC.en ← 禁止
```

			IF级	ID级	EX级	MEM级	WB级		
地址	指令	CLK	PC	IM	IF/ID	ID/EX	EX/MEM	MEM/WB	RF
0	lw $t0, 0($t1)	↑ 1	0→4	lw→sub	lw				
4	sub $t3, $t0, $t2	↑ 2	4→8	sub→and	sub	lw			
8	and $t5, $t0, $t4	↑ 3	8→8	and	sub	nop	lw		
12	or $t7, $t0, $t6								
16	add $t1, $t2, $t3								

如果没有转发电路呢？

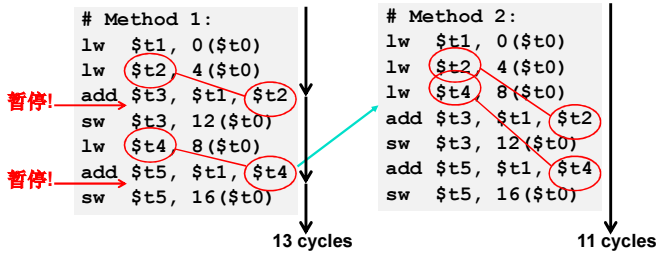
- 由于有转发电路，因此lw指令只插入1个NOP指令
- Q：如果没有转发，需要怎么处理呢？
- A：EX/MEM，MEM/WB也均需要做冲突分析及NOP处理
  - EX/MEM，MEM/WB也需要修改，并增加相应控制信号

			IF级	ID级	EX级	MEM级	WB级		
地址	指令	CLK	PC	IM	IF/ID	ID/EX	EX/MEM	MEM/WB	RF
0	lw $t0, 0($t1)	↑ 1	0→4	lw→sub	lw				
4	sub $t3, $t0, $t2	↑ 2	4→8	sub→and	sub	lw			
8	and $t5, $t0, $t4	↑ 3	8	and	sub	nop	lw		
12	or $t7, $t0, $t6	↑ 4	8	and	sub	nop	nop	lw结果	
16	add $t1, $t2, $t3	↑ 5	8	and	sub	nop	nop	nop	lw结果
		↑ 6	8→12	and→or	and	sub	nop	nop	nop

- Load之后的时间片段称为load延迟时隙
  - 如果后续指令会用到load回来的结果，硬件互锁机制会暂停该指令一个时钟周期
  - 在延迟时隙由硬件暂停指令等价于在该时隙加nop (除非加nop占用更多的代码空间)
- Idea: 让编译器在该时隙插入一条不相关的指令 → 无需暂停!

避免暂停指令的代码调度

- 对代码重排序以避免下一条指令使用load的结果!
- MIPS 代码：A=B+E； C=B+F；



控制冒险

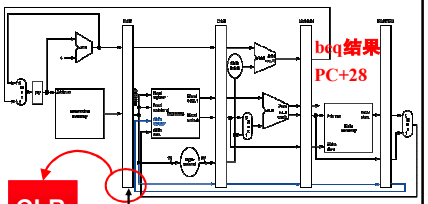
- 分支 (beq, bne) 决定控制流
  - 下一条指令的获取依赖于分支的结果
  - 流水线无法总是取到正确的指令
    - 仍然会执行ID阶段
- 简单的解决方案：暂停每一个分支直到获得新的PC值
  - 我们必须暂停多长时间?

分支指令冒险造成的停顿代价

地址	指令	CLK	PC	IM	IF/ID	ID/EX	EX/MEM	MEM/WB	RF
					IF级	ID级	EX级	MEM级	WB级
0	beq $1, $3, 24	↑ 1	0→4	beq→and	beq				
4	and $12, $2, $5	↑ 2	4	and	nop	beq			
8	or $13, $6, $2	↑ 3	4	and	nop	nop	beq结果		
12	add $14, $2, $2	↑ 4	4→28	and→lw	nop	nop	nop		
		↑ 5	28→32	lw→XX	lw	nop	nop	nop	nop
28	lw $4, 50($7)								

- 如不对分支指令做任何处理，则必须插入3个NOP
  - 分支指令结果及新PC值保存在EX/MEM，因此PC在clk4才能加载正确值
  - IF/ID在clk5才能存入转移后指令(即lw指令)

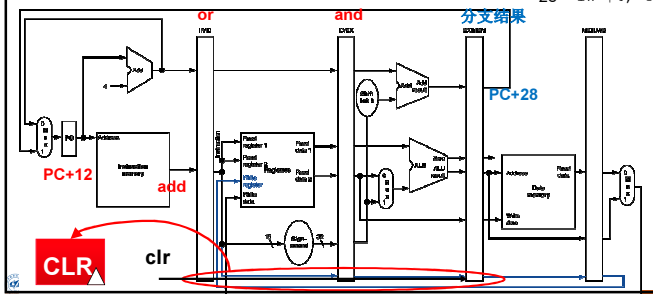
Q：IF/ID.clr表达式?



方案1：假定分支不发生

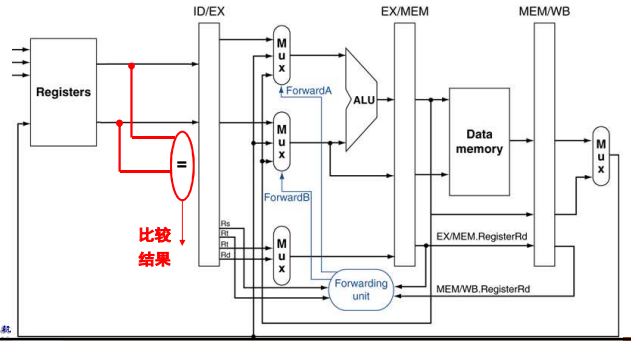
- 即使在ID级发现是分支指令也不停顿
- 根据分支指令结果，决定是否清除3条后继指令
  - 使得and/or/add不能前进

PC相对偏移	指令
0	beq $1, $3, 24
4	and $12, $2, $5
8	or $13, $6, $2
12	add $14, $2, $2
28	lw $4, 50($7)



方案2：缩短分支延迟

- 在ID阶段放置比较器，尽快得到分支指令结果
  - 分支指令结果可以提前2个clock得到
  - 分支指令后继可能被废弃的指令减少为1条
    - 当需要转移时，清除IF/ID即可

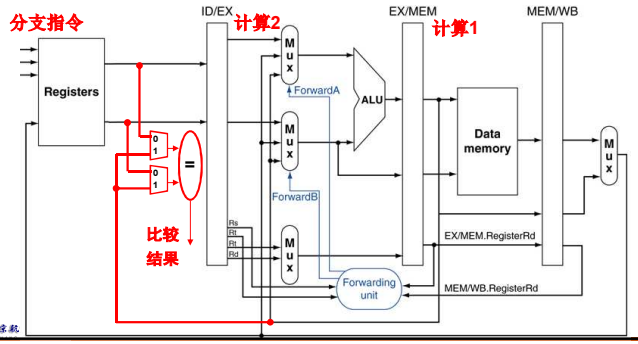


方案2：缩短分支延迟

- 比较器前置后，会产生数据相关
  - 分支指令可能依赖于前条指令的结果
- 依赖计算1：从ALU转发数据
- 依赖计算2：只能暂停

Q：如果依赖MEM/WB的结果，是否需要设置转发？

提示：MEM/WB已经有回写通道了，但RF设计满足吗？



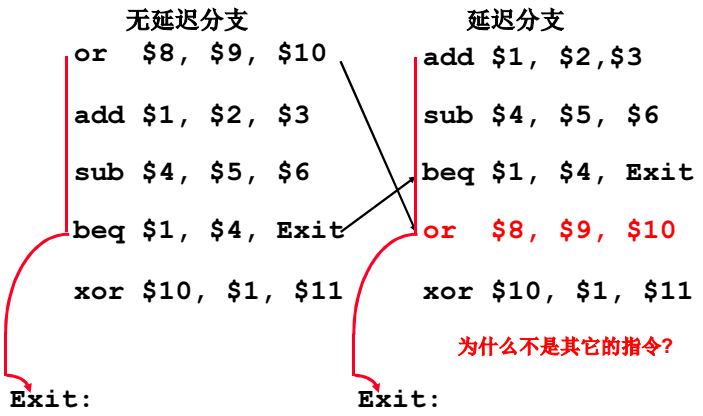
控制冒险: 分支

- 选择 #3: 分支延迟时隙
  - 无论是否分支转跳，总是立即执行分支之后的指令
  - 最坏情况: 增加一个nop 分支延迟时隙
  - 最好情况: 移动一条指令放入分支延迟时隙
    - 必须不影响程序的逻辑

控制冒险: 分支

- MIPS 使用延迟的分支这一概念
  - 对指令重排序是加速程序执行的常用方法
  - 编译器选择一条指令放入分支延迟时隙执行大约能节省50%的时间
- Jump指令也有延迟时隙
  - 为什么需要?

延迟分支的例子



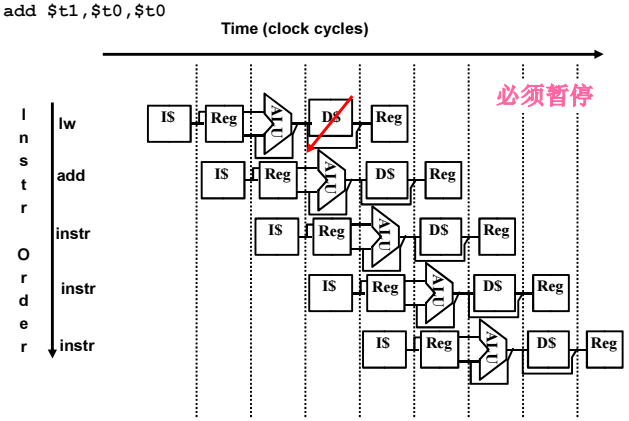
延迟的Jump

❖**MIPS Green Sheet**

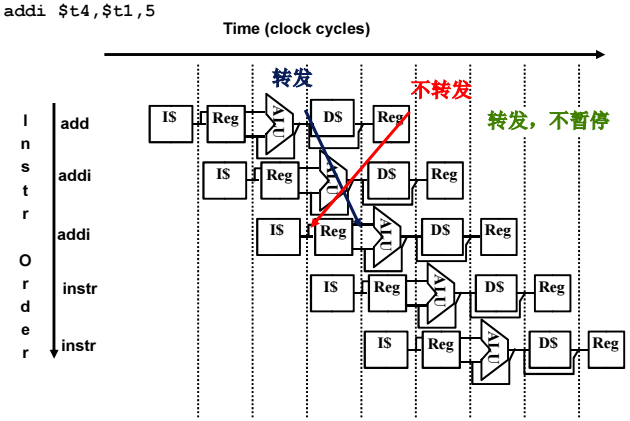
```
jal:
R[31]=PC+8;  PC=JumpAddr
```

- PC+8 就是因为*jump 延迟时隙*!
- PC+4所指的指令总是在jal转跳向label之前被执行，所以返回PC+8


```
lw   $t0,0($t0)
add $t1,$t0,$t0
```




```
add $t1,$t0,$t0
addi $t2,$t0,5
addi $t4,$t1,5
```




```
addi $t1,$t0,1
addi $t2,$t0,2
addi $t3,$t0,2
addi $t3,$t0,4
addi $t5,$t1,5
```